Model-Based Testing with Graph Grammars

MSc Thesis *(Afstudeerscriptie)*

written by

Vincent de Bruijn

Formal Methods & Tools,
University of Twente,
Enschede,
The Netherlands

v.debruijn@student.utwente.nl

August 24, 2012

Abstract

Graph Grammars have many structural advantages, which are potential benefits for the model-based testing process. We describe a model-based testing setup with Graph Grammars. The result is a system for automatic test generation from Graph Grammars. A graph transformation tool, GROOVE, and a model-based testing tool, ATM, are used as the backbone of the system. The system is validated using the results of several case-studies.

Contents

Chapter 1

Introduction

In software development projects, often time and budget costs are exceeded. Laird and Brennan [11] investigated in 2006 that 23% of all software projects are canceled before completion. Furthermore, of the completed projects, only 28% are delivered on time with the average project overrunning the budget with 45%. Testing is an important part of software development, because it decreases future maintainance costs [17]. Testing is a complex process and should be done often [20]. Therefore, the testing process should be as efficient as possible in order to save resources.

Test automation allows repeated testing during the development process. The advantage of this is that bugs are found early and can therefore be fixed early. A widely used practice is maintaining a *test suite*, which is a collection of test-cases. However, when the creation of a test suite is done manually, this still leaves room for human error [14]. The process of deriving tests tends to be unstructured, barely motivated in the details, not reproducible, not documented, and bound to the ingenuity of single engineers [28].

1.1 Model-based Testing

The existence of an artifact that explicitly encodes the intended behaviour can help mitigate the implications of these problems. Creating an abstract representation or a *model* of the system is an example of such an artifact. What is meant by a model in this report, is the description of the behavior of a system. Moreover, the term model will be often used to describe transition-based notations, such as finite state machines, labelled transition systems and I/O automata. Statecharts such as UML models are not considered in this report.

A model can be used to systematically generate tests for the system. This is referred to as *model-based testing*. This leads to a larger test suite in a shorter amount of time than if done manually. These models are created from the specification documents provided by the end-user. These specification documents are 'notoriously error-prone' [16]. This implies that the model itself needs validation. Validating the model usually means that the requirements themselves are scrutinised for consistency and completeness [28].

Tools for automatic test generation already exist. The testing tool developed by Axini[1] is used for the automatic test generation on *symbolic* models, which combine a state and data type oriented approach. This tool is used in this report and is referred to as Axini Test Manager (ATM). In Utting et al. [28], a taxonomy is done on different model-based testing tools:

[1] http://www.axini.nl/

- TorX [25]: accepts behaviour models such as I/O labelled transition systems. A version of this tool written in Java under continuous development is JTorX [2].
- Spec Explorer[29]: provides a model editing, composition, exploration, and visualization environment within Visual Studio, and can generate offline .NET test suites or execute tests as they are generated (online).
- JUMBL[21]: an academic model-based statistical testing that supports the development of statistical usage-based models using Markov chains, the analysis of models, and the generation of test cases.
- AETG[5]: implements combinatorial testing, where the number of possible combinations of input variables are reduced to a few 'representative' ones.

The stakeholders evaluate the constructed model to verify its correctness. However, the visual or textual representation of large models may become troublesome to understand, which is referred to as the model having a low model transparency or high model complexity. Models that are often used are transition-based, i.e. a collection of nodes representing the states of the system connected by transitions representing an action taken by the system. The problem with such models is that a larger number of states decreases the model transparency. We think that low model transparency make errors harder to detect and that it obstructs the feedback process of the stakeholders. Using models with high transparency is therefore essential.

1.2 Graph Transformation

A formalism that claims to have more model transparency is Graph Transformation. The system states are represented by graphs and the transitions between the states are accomplished by applying graph change rules to those graphs. These rules can be expressed as graphs themselves. A graph transformation model of a software system is therefore a collection of graphs, each a visual representation of one aspect of the system. This formalism may therefore provide a more intuitive approach to system modelling than traditional state machines. Graph Transformation and its potential benefits have been studied since the early '70s. The usage of this computational paradigm is best described by the following quote from Andries et al. [1]:

> Graphs are well-known, well-understood, and frequently used means to represent system states, complex objects, diagrams, and networks, like flowcharts, entity-relationship diagrams, Petri nets, and many more. Rules have proved to be extremely useful for describing computations by local transformations: Arithmetic, syntactic, and deduction rules are well-known examples.

An informative paper on graph transformations is written by Heckel et al. [9]. A quote from this paper:

> Graphs and diagrams provide a simple and powerful approach to a variety of problems that are typical to computer science in general, and software engineering in particular.

The graph transformation tool GROOVE[2] is used to model and explore graph grammars.

1.3 Research goals

The motivation above is given for using graph grammars as a modelling technique. The goal of this research is to create a system for automatic test generation on graph grammars. If the assumptions that graph grammars provide a more intuitive modelling and testing process hold,

[2] http://sourceforge.net/projects/groove/

this new testing approach will lead to a more efficient testing process and fewer incorrect models. The to be designed system, once implemented and validated, provides a valuable contribution to the testing paradigm. The tools GROOVE and ATM are used to create this system.

The research goals are split into a design and validation component:

1. **Design**: Design and implement a system using ATM and GROOVE which performs model-based testing on graph grammars.
2. **Validation**: Validate the design and implementation using case studies and performance measurements.

The result of the design goal is one system called the GROOVE-Axini Testing System (GRATiS). The validation goal uses case-studies with existing specifications from systems tested by Axini. Each case-study has a graph grammar and a symbolic model which describe the same system. GRATiS and ATM are used for the automatic test generation on these models respectively. Both the models and the test processes are compared as part of the validation.

The solution has to uphold three requirements:

1. A graph grammar must be used as the model; it must derive from the specification and be used for the testing.
2. It must be possible to measure the test progress/completion, by means of *coverage* statistics (explained in detail in section 2.1.4).
3. The solution must be efficient: it should be usable in practice, therefore the technique should be scalable and the imposed constraints reasonable from a practical view point.

1.4 Roadmap

This report features five more chapters: first, the concepts described in this chapter are elaborated in chapter 2. The technique used in GRATiS is described in chapter 3. The setup for the validation of GRATiS is in chapter 5. Chapter 6 features model examples on which GRATiS is applied and validated. Chapter 7 features a case study where GRATiS is applied on an existing software system. Finally, conclusions are drawn in chapter 8.

Chapter 2

Background

The structure of the rest of this chapter is as follows: the general model-based testing process is set out in section 2.1. Some basic concepts from algebra are described in section 2.2. The symbolic models from ATM are then described in section 2.3. Section 2.4 describes the graph grammar formalism. GROOVE and ATM are described in section 2.5.

2.1 Model-based Testing

Model-based testing is a testing technique where a System Under Test (SUT) is tested for conformance to a model description of the system. The general setup for this process is depicted in Figure 2.1. The specification of a system, given as a model, is given to a test derivation component which generates test cases. These test cases are passed to a component that executes the test cases on the SUT. Tests are executed by providing input/stimuli to the SUT and monitoring the output/response. The test execution component evaluates the test cases, the stimuli and the responses. It gives a 'pass' or 'fail' verdict depending on whether the SUT conforms to the specification or not respectively.

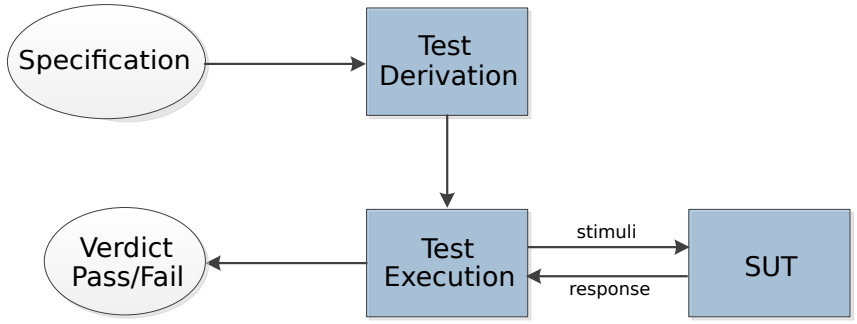


Figure 2.1: A general model-based testing setup

This type of model-based testing is called *batch testing* or *offline testing*. Another type of model-based testing is *on-the-fly* testing. The main difference is that no test cases are derived, instead a transition in the model is chosen and tested on the system directly. The general architecture for this process is shown in Figure 2.2. An example of an on-the-fly testing tool is TorX [25].

Variations of state machines and transition systems have been widely used as the underlying model for test generation. Other tools use the structure of data types to generate test data.

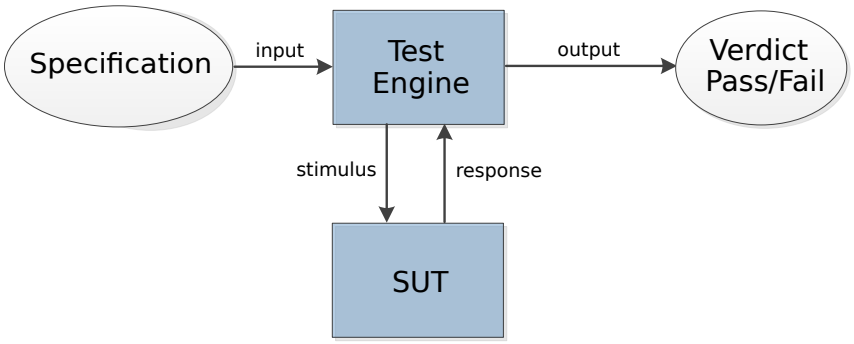


Figure 2.2: A general 'on-the-fly' model-based testing setup

The stucture of the rest of this section is as follows. First, previous work on model-based testing is given. Then, two types of models are introduced. These are basic formalisms useful to understand the models in the rest of the paper. Finally, the notion of *coverage* is explained.

2.1.1 Previous work

Formal testing theory was introduced by De Nicola et al. [19]. The input-output behavior of processes is investigated by series of tests. Two processes are considered equivalent if they pass exactly the same set of tests. This testing theory was first used in algorithms for automatic test generation by Brinksma [3]. This led to the so-called *canonical tester* theory. Tretmans gives a formal approach to protocol conformance testing (whether a protocol conforms to its specifications) in [26] and an algorithm for deriving a sound and exhaustive test suite from a specification in [27]. A good overview of model-based testing theory and past research is given in "Model-Based Testing of Reactive Systems" [15].

2.1.2 Labelled Transition Systems

A labelled transition system is a structure consisting of states with labelled transitions between them.

Definition 2.1.1. A labelled transition system is a 4-tuple $\langle Q, L, T, q_0 \rangle$, where:

- Q is a finite, non-empty set of states
- L is a finite set of labels
- $T \in Q \times (L \cup \{\tau\}) \times Q$, with $\tau \notin L$, is the transition relation
- $q_0 \in Q$ is the initial state.

We write $q \xrightarrow{\mu} q'$ if there is a transition labelled μ from state q to state q', i.e., $(q, \mu, q') \in T$. The informal idea of such a transition is that when the system is in state q it may perform action μ, and go to state q'.

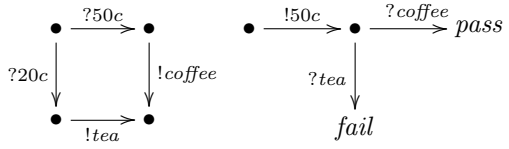
2.1.3 Input-Output Transition Systems

A useful type of transition system for model-based testing is the Input-Output Transition System (IOTS) by Tretmans [27]. Assuming that implementations communicate with their environment via inputs and outputs, this formalism is useful for describing system behavior. IOTSs have the same definition as LTSs with one addition: each label $l \in L$ has a type $\iota \in Y$, where $Y =$

$\{input, output\}$. Each label can therefore specify whether the action represented by the label is a possible input or an expected output of the system under test.

An example of such an IOTS is shown in Figure 2.3a. This system allows an input of 20 or 50 cents and then outputs tea or coffee accordingly. The inputs are preceded by a '?', the outputs are preceded by an '!'. This system is a specification of a coffee machine. A test case can also be described by an IOTS with special pass and fail states. A test case for the coffee machine is given in Figure 2.3b. The test case shows that when an input of '50c' is done, an output of 'coffee' is expected from the tested system, as this results in a 'pass' verdict. When the system responds with 'tea', the test case results in a 'fail' verdict. The pass and fail verdicts are two special states in the test case, which are sink states, i.e., once in either of those the test case cannot leave that state.

Test cases should always reach a pass or fail state within finite time. This requirement ensures that the testing process halts.



(a) An IOTS (b) An IOTS test case

Figure 2.3: The specification of a coffee machine and a test case as an IOTS

2.1.4 Coverage

The number of tests that can be generated from a model is potentially infinite. Therefore, there must be a test selection strategy to maximize the quality of the tests while minimizing the time spent testing. Coverage statistics help with test selection. Such statistics indicate how much of the SUT is tested. When the SUT is a black-box, typical coverage metrics are state and transition coverage of the model [13, 18, 8].

As an example, let us calculate the coverage metrics of the IOTS test case example in 2.3b. The test case tests one path through the specification and passes through 3 out of 4 states and 2 out of 4 transitions. The state coverage is therefore 75% and the transition coverage is 50%.

Coverage statistics are calculated to indicate how adequately the testing has been performed [30]. These statisics are therefore useful metrics for communicating how much of a system is tested.

2.2 Algebra

Some basic concepts from algebra are described here. For a general introduction into logic we refer to [10].

A *multi-sorted signature* $\langle S, F \rangle$ describes the function symbols and sorts of a formal language. F is a set of function symbols. S is a set of sorts. Each $f \in F$ has an arity $n \in \mathbb{N}$, where a function symbol with arity $n = 0$ is called a constant symbol. F^i denotes the subset of F, with function symbols of arity $n = i$. The sort of a function symbol $f \in F$ with arity n is given by $\sigma(f) = s_1 ... s_n + 1$, with $s_i \in S$ for $1 \leq i \leq n$. S_{n+1} is the return sort. In this report, $S = \{int, real, bool, string\}$ denoting the integer, real, boolean and string sorts respectively. F

features the commonly used function symbols, which include, but not restricted by, '+', '*', '=', '<', '0', '1'.

An *algebra* $\mathcal{A} = \langle \mathbb{U}, \Phi \rangle$ has a non-empty set $\mathbb{U}$ of constants called a *universe*, partitioned into $\mathbb{U}^s$ for each $s \in S$, and a set Φ of functions. A function $\phi_{\mathcal{A}}$ is typed $\mathbb{U}_{\mathcal{A}}^{s_1} \times ... \mathbb{U}_{\mathcal{A}}^{s_n} \rightarrow \mathbb{U}_{\mathcal{A}}^{s_{n+1}}$, where $s_1 ... s_{n+1}$ is the sort of the function symbol given by the signature. For example, $<_{\mathcal{A}}$: $\mathbb{U}_{\mathcal{A}}^{int} \times \mathbb{U}_{\mathcal{A}}^{int} \rightarrow \mathbb{U}_{\mathcal{A}}^{bool}$ represents the 'less-than' comparison of two integers.

We define $\mathcal{V} = \mathcal{V}^{int} \uplus \mathcal{V}^{real} \uplus \mathcal{V}^{bool} \uplus \mathcal{V}^{string}$ to be the set of *variables*. *Terms* over V, denoted $\mathcal{T}(V)$, are built from function symbols F and variables $V \subseteq \mathcal{V}$. The definition of a term is:

$$\begin{array}{rcll} t & ::= & f(t_1 ... t_n) & \\ & | & x & , \textit{ where } x \textit{ is a constant.} \end{array}$$

We write $var(t)$ to denote the set of variables appearing in a term $t \in \mathcal{T}(V)$. Terms $t \in \mathcal{T}(\emptyset)$ are called ground terms. An example of a term t is $(x + (y - 1))$, with $var(t) = \{x, y\}$. The type of a term is given by:

$$\begin{array}{rll} \sigma : t \mapsto & s & \textit{if } t = x \in \mathcal{V}^s \\ & s_{n+1} & \textit{if } t = \phi(t_1 ... t_n) \textit{ and } \sigma(\phi) = s_1 ... s_{n+1}, \textit{ provided } \sigma(t_i) = s_i \end{array}$$

The set of terms with return types $\mathbb{U}^{bool}$, is denoted as $\mathcal{B}(\mathcal{V})$. An example is $(x < y)$, where the result is *true* or *false*.

A *term-mapping* is a function $\mu : \mathcal{V} \rightarrow \mathcal{T}(\mathcal{V})$. A *valuation* ν is a function $\nu : \mathcal{V} \rightarrow \mathbb{U}$ that assigns values to variables. For example, given an algebra, $\nu : \{(x \mapsto 1), (y \mapsto 2))\}$ assigns the values 1 and 2 to the variables x and y respectively. A valuation of a term given $\mathcal{A}$ is defined by:

$$\begin{array}{rlcl} \nu : & x & \mapsto & \nu(x) \\ & (f(t_1 ... t_n)) & \mapsto & f_{\mathcal{A}}(\nu(t_1) ... \nu(t_n)) \end{array}$$

When every variable in a term is defined by a valuation, the term can be valuated to a value. Therefore, when every variable in a term-mapping is defined by a valuation, a new valuation can be obtained. Formally, this is defined as: $\_after\_ : (\mathcal{V} \rightarrow \mathbb{U}) \times (\mathcal{V} \rightarrow \mathcal{T}(\mathcal{V})) \rightarrow (\mathcal{V} \rightarrow \mathbb{U})$. Given a valuation ν and a term-mapping μ, $(\nu \textit{ after } \mu) : \nu \mapsto \nu(\mu(\nu))$.

2.3 Symbolic Transition Systems

Symbolic Transition Systems (STSs) combine a state oriented and data type oriented approach. These systems are used in practice in ATM and will therefore be part of GRATiS. In this section, previous work on STSs is given. The definitions of STSs and IOSTSs follow. An example of an IOSTS is then given. Next, the transformation of an STS to an LTS is explained and illustrated by an example. This transformation is useful when comparing STSs to systems that are not STSs. Finally, different coverage metrics on STSs are explained.

2.3.1 Previous work

STSs are introduced by Frantzen et al. [12]. This paper includes a detailed definition, on which the definition in section 2.3.2 is based. The authors also give a sound and complete test derivation algorithm from specifications expressed as STSs. Deriving tests from a symbolic specification or *Symbolic test generation* is introduced by Rusu et al. [24]. Here, the authors use *Input-Output Symbolic Transition Systems* (IOSTSs). These systems are very similar to the STSs in [12]. However, the definition of IOSTSs we will use in this report is based on the STSs by [12]. A tool that generates tests based on symbolic specifications is the STG tool, described in Clarke et al. [4].

2.3.2 Definition

An STS has *locations* and *switch relations*. If the STS represents a model of a software system, a location in the STS represents a state of the system, not including data values. A switch relation defines the transition from one location to another. The *location variables* are a representation of the data values in the system. A switch relation has a *gate*, which is a label representating the execution steps of the system. Gates have *interaction variables*, which represent some input or output data value. Switch relations also have *guards* and *update mappings*. A guard is a term $t \in \mathcal{B}(\mathcal{V})$. The guard disallows using the switch relation when the valuation of the term results in *false*. When the valuation results in *true*, the switch relation of the guard is *enabled*. An update mapping is a term-mapping of location variables. After the system switches to a new location, the variables in the update mapping will have the value corresponding to the valuation of the term.

Definition 2.3.1. A Symbolic Transition System is a tuple $\langle W, w_0, \mathcal{L}, \imath, \mathcal{I}, \Lambda, D \rangle$, where:

- W is a finite set of locations and $w_0 \in W$ is the initial location.
- $\mathcal{L} \subseteq \mathcal{V}$ is a finite set of location variables.
- $\imath$ is a term-mapping $\mathcal{L} \to \mathcal{T}(\emptyset)$, representing the initialisation of the location variables.
- $\mathcal{I} \subseteq \mathcal{V}$ is a set of interaction variables, disjoint from $\mathcal{L}$.
- Λ is a finite set of gates. The unobservable gate is denoted $\tau (\tau \notin \Lambda)$; we write Λ_τ for $\Lambda \cup \{\tau\}$. The arity of a gate $\Lambda \in \Lambda_\tau$, denoted $arity(\Lambda)$, is a natural number. The parameters of a gate $\Lambda \in \Lambda_\tau$, denoted $param(\Lambda)$, are a tuple of length $arity(\Lambda)$ of distinct interaction variables. We fix arity$(\tau) = 0$, i.e. the unobservable gate has no interaction variables.
- $D \subseteq W \times \Lambda_\tau \times \mathcal{B}(\mathcal{L} \cup \mathcal{I}) \times (\mathcal{L} \to \mathcal{T}(\mathcal{L} \cup \mathcal{I})) \times W$, is the switch relation. We write $w \xrightarrow{\lambda,\gamma,\rho} w'$ instead of $(w, \lambda, \gamma, \rho, w') \in D$, where γ is referred to as the guard and ρ as the update mapping. We require $var(\gamma) \cup var(\rho \subseteq \mathcal{L} \cup param(\lambda))$. We define $out(w) \subseteq D$ to be the outgoing switch relations from location w.

2.3.3 Input-Output Symbolic Transition Systems

An IOSTS can now easily be defined. The same difference between LTSs and IOTSs applies, namely each gate in an IOSTS has a type $\iota \in Y$. As with IOTSs, each gate is preceded by a '?' or '!' to indicate whether it is an input or an output respectively.

2.3.4 Example

In Figure 2.4 the IOSTS of a simple board game is shown, where two players consecutively throw a die and move along four squares. The 'init' switch relation is a graphical representation of the variable initialization $\imath$. The values in the tuple of the IOSTS are defined as follows:

$$
\begin{array}{lcl}
W & = & \{t, m\} \\
w_0 & = & t \\
\mathcal{L} & = & \{T, P1, P2, D\} \\
\imath & = & \{T \mapsto 0, P1 \mapsto 0, P2 \mapsto 2, D \mapsto 0\} \\
\mathcal{I} & = & \{d, p, l\} \\
\Lambda & = & \{?throw, !move\} \\
D & = & \{t \xrightarrow{?throw, 1<=d<=6, D \mapsto d} m, \\
 & & \quad m \xrightarrow{!move, T=1 \wedge l=(P1+D)\%4, P1 \mapsto l, T \mapsto 2} t, \\
 & & \quad m \xrightarrow{!move, T=2 \wedge l=(P2+D)\%4, P2 \mapsto l, T \mapsto 1} t\}
\end{array}
$$

The variables $T, P1, P2$ and D are the location variables symbolizing the player's turn, the positions of the players and the number of the die thrown respectively. The output gate $!move$ has $param = \langle p, l \rangle$ symbolizing which player moves to which location. The input gate $?throws$ has $param = \langle d \rangle$ symbolizing which number is thrown by the die. The switch relation with gate $?throws$ has the restriction that the number of the die thrown is between one and six and the update sets the location variable D to the value of interaction variable d. The switch relations with gate $!move$ have the restriction that it must be the turn of the player moving and that the new location of the player is the number of steps ahead as thrown by the die. The update mapping sets the location of the player to the correct value and passes the turn to the next player. In Figure 2.4 the gates, guards and updates are separated by pipe symbols '|' respectively.

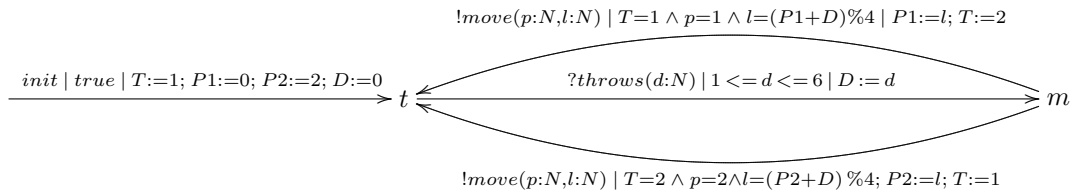


Figure 2.4: The STS of a board game example

2.3.5 STS to LTS mapping

Consider an STS J and an LTS K. There exists a mapping from the location and location variable valuations to the states of K and from the switch relations and variable valuations of J to the transitions of K, such that K is an expansion of J. These relations are defined as follows:

$\mu_Q: \quad (W \times (\mathcal{L} \to \mathbb{U})) \to Q$

$\mu_L: \quad (\Lambda \times (\mathcal{I} \to \mathbb{U})) \to L$

$\mu_T: \quad (w \xrightarrow{\lambda,\gamma,\rho} w', \nu : ((\mathcal{L} \cup \mathcal{I}) \to \mathbb{U})) \mapsto (\mu_Q(w, \nu \restriction \mathcal{L}) \xrightarrow{\mu_L(\lambda,\nu\restriction\mathcal{I})} \mu_Q(w', \nu \; after \; \rho))$

When the number of possible valuations for $\mathcal{L}$ and $\mathcal{I}$ and the number of locations in an STS is considered to be finite, the transformation is always possible to an LTS with finite number of states.

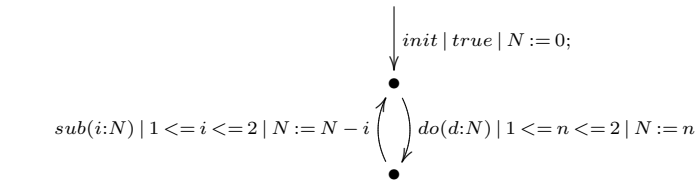
An example of this transformation is shown in Figure 2.5. The label 'do(1)' in the LTS is a textual representation of the gate 'do' plus a valuation of the interaction variable 'd'. The text on the nodes indicate from which location and valuation the state was created. The node labelled '$w_0, N = 2$' is an example of an unreachable state.

2.3.6 Coverage

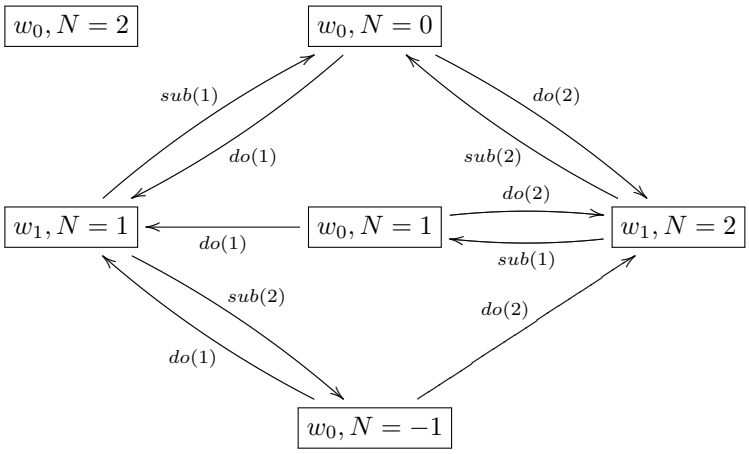
The simplest metric to describe the coverage of an STS is the location and switch-relation coverage, which express the percentage of locations and switch relations tested in the test run. Measuring state and transition coverage of an STS is possible using the LTS resulting from the STS transformation.

2.4 Graph Grammars

A *Graph Grammar* (GG) is composed of a set of graph transformation rules. These rules indicate how a graph can be transformed to a new graph. These graphs are called *host graphs*. The rules are composed of graphs themselves, which are called *rule graphs*.



(a) The STS



(b) The LTS

Figure 2.5: An example of a transformation of an STS to an LTS

The rest of this section is ordered as follows: first, graphs, host graphs, rule graphs and graph transformation rules are explained. Then, the definition of a *Graph Transition System* (GTS) is given. An example of a GG and a GTS is then given. Finally, the definition of IOGGs is given. For a more detailed overview of GGs, we refer to [22, 9, 1].

Definition 2.4.1. A *graph* is composed of nodes and edges. In this report, we assume a universe of nodes $\mathbb{V} = \mathbb{W} \uplus \mathbb{U} \uplus \mathcal{V} \uplus 2^{\mathcal{T}}$, where $\mathbb{W}$ is the universe of standard graph nodes. $\mathbb{E}$ is the universe of edges between two nodes in $\mathbb{V}$.

Definition 2.4.2. A host graph G is a tuple $\langle V_{G^h}, E_{G^h} \rangle$, where:

- $V_{G^h} \subseteq (\mathbb{W} \uplus \mathbb{U})$ is the node set of G
- $E_{G^h} \subseteq (V_{G^h} \backslash \mathbb{U} \times L \times V_{G^h})$ is the edge set of G

Figure 2.6 shows an example of a host graph. Here, $n_1, n_2 \in \mathbb{W}$ are the *identities* of the nodes. The other four nodes are values in $\mathbb{U}$.

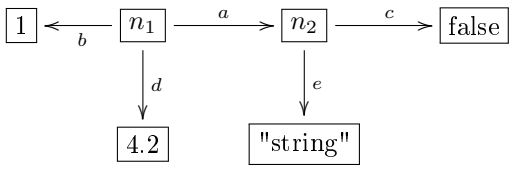


Figure 2.6: An example of a host graph

Definition 2.4.3. A rule graph H is a tuple $\langle V_{G^r}, E_{G^r} \rangle$, where:

- $V_{G^r} \subseteq (\mathbb{V} \backslash \mathbb{U})$ is the node set of H

- $E_{G^r} \subseteq (V_{G^r} \times L \times V_{G^r})$ is the edge set of H

In addition, the following must hold:

- $\forall z \in V_{G^r} \wedge z \in 2^{\mathcal{T}} \,.\, var(z) \subseteq V_{G^r}$ - The variables used in the terms must be present as nodes in the rule graph.
- $(\forall z \in V_{G^r} \wedge z \in \mathcal{V} \,.\, \exists(\_, \_, z) \in E_{G^r}$ - If a variable is used in a rule graph, it needs context. Therefore, there must be an edge with the variable node as target.

Figure 2.7 shows an example of a rule graph. Here, $r_1, r_2 \in \mathbb{W}$ are the node identities, $x_1, x_2 \in \mathcal{V}^{int}$ and $\{x_1 + 1, x_2\} \in 2^{\mathcal{T}}$. The set of terms is mapped as a node to the same value. This mapping is explained in the next definition. The consequence is that this node implicitly expresses the relation $x_1 + 1 = x_2$.

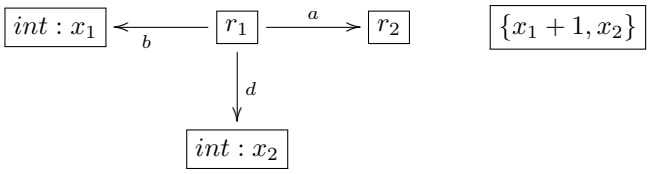


Figure 2.7: An example of a rule graph

Definition 2.4.4. A graph g has a *morphism* to a graph g' if there is a structure-preserving mapping from the nodes and the edges of g to the nodes and the edges of g' respectively. A graph g has a partial morphism to a graph g' if there are elements in g without an image in g'.

Definition 2.4.5. A node or edge z in graph g has an *image* in graph g' and z is a *pre-image* of the image. For each type of node, an explanation is given A variable $v \in \mathcal{V}^s, s \in S$, has an image i in a host graph if $i \in \mathbb{U}^s$. A node $z \in 2^{\mathcal{T}}$ has an image i in a host graph if i is the valuation of all terms in z.

Definition 2.4.6. A transformation rule is a tuple $\langle LHS, NAC, RHS, l \rangle$, where:

- *LHS* is a rule graph representing the left-hand side of the rule
- *NAC* is a set of rule graphs representing the negative application conditions
- *RHS* is a rule graph representing the right-hand side of the rule
- $l \in L$ is the label of the rule

There exist implicit partial morphisms from the *LHS* to each rule graph in *NAC* and from the *LHS* to the *RHS* by means of the node identities. These morphisms are *rule graph morphisms*.

Definition 2.4.7. A *creator* edge is an edge in the *RHS* of a rule, but not in the *LHS* of the rule. An *eraser* edge is an edge in the *LHS* of a rule, but not in the *RHS* or a rule.

Definition 2.4.8. A rule r has a *rule match* on a host graph G if its *LHS* has a morphism in G and $\nexists n \in NAC$ such that n has a morphism in G and $\forall e \in LHS$, if e has an image i in n, and an image j in G, then j should be an image of i. The morphism of the *LHS* to a host graph is a *match morphism*.

Definition 2.4.9. After the rule match is applied to the graph, all elements in *LHS* that do not have an image in *RHS*, are removed from G and all elements in *RHS* that do not have a pre-image in *LHS*, are added to G. This process, called a *rule transition*, is denoted $G \xrightarrow{r,m} G'$, where $m \in M$ is the morphism of the *LHS* to G.

Figure 2.8 shows an example of the initial graph G_0, one rule of a GG and the corresponding rule match. G_0 can be represented by $\langle \{n1, n2\}, \{\langle n1, a, n1\rangle, \langle n1, A, n2\rangle, \langle n2, B, n2\rangle\}\rangle$. The *LHS* of

the rule has a match in G_0. Neither *NAC1* and *NAC2* have a match in G_0, because the edge with label C does not exist in G_0. The new graph after applying the rule is G_1.

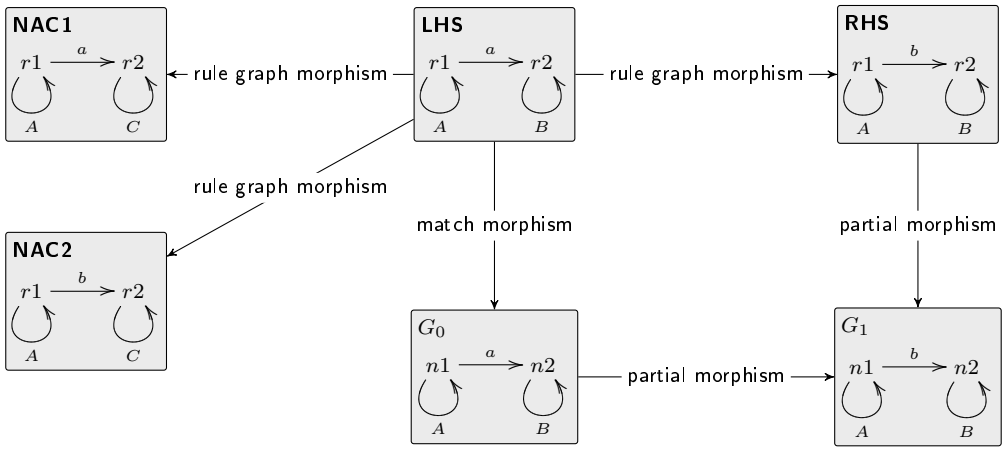


Figure 2.8: An example of a graph transformation

Definition 2.4.10. A graph grammar is a tuple $\langle R, G_0 \rangle$, where:

- R is a set of graph transformation rules
- G_0 is the initial graph

By repeatedly applying graph transformation rules to the start graph and all its consecutive graphs, a GG can be explored to reveal a *Graph Transition System* (GTS). This transition system consists of graphs connected by rule transitions.

Definition 2.4.11. A graph transition system is a tuple $\langle \mathcal{G}, R, M, U, G_0 \rangle$, where:

- $\mathcal{G}$ is a set of graphs
- $L \in R \times M$ is a set of labels
- $U \in \mathcal{G} \times L \times \mathcal{G}$ is the rule transition relation
- $G_0 \in \mathcal{G}$ is the initial graph

Let $K = \langle R, G_0$. A GTS $O = \langle \mathcal{G}, R, M, U \rangle$ is derived from a K by the following. $\mathcal{G}, M, U$ are the smallest sets, such that:

- $G_0 \in \mathcal{G}$
- if $G \in \mathcal{G}$ and $G \xrightarrow{r,m} G'$ then $G' \in \mathcal{G}, (r, m) \in L, (G \xrightarrow{r,m} G') \in U$

Definition 2.4.12. In order to specify stimuli and responses with GGs, a definition is given for an *Input-Output GG* (IOGG). Concretely, the IOGG places input and output labels on its rule transitions. Following the definition from IOLTSs, each rule label $l \in L$ has a type $\iota \in Y$. Exploring an IOGG leads to an *Input-Output Graph Transition System* (IOGTS). The rule transitions derive their type from their corresponding rule.

2.5 Tooling

2.5.1 ATM

ATM is a model-based testing web application, developed in the Ruby on Rails framework. It is used to test the software of several big companies in the Netherlands since 2006. It is under continuous development by Axini.

The architecture is shown graphically in Figure 2.9. It has a similar structure to the on-the-fly model-based testing tool architecture in Figure 2.2.

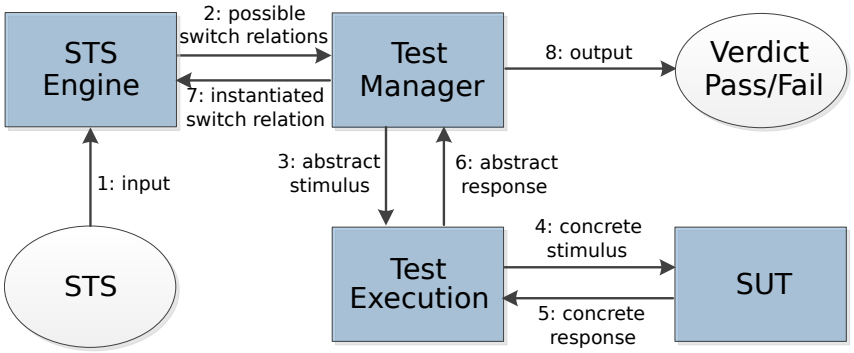


Figure 2.9: Architecture of ATM

The tool functions as follows:

1. An STS is given to an STS Engine, which keeps track of the current location and data values. It passes the possible switch relations from the current location to the Test Manager.
2. The Test Manager chooses an enabled switch relation based on a test strategy, which can be a random strategy or a strategy designed to obtain a high location/switch relation coverage. The valuation of the variables in the guard are also chosen by a test strategy, which can be a random strategy or a strategy using boundary-value analysis. The choice is represented by an instantiated switch relation and passed back to the STS Engine, which updates its current location and data values. The communication between these two components is done by method calls.
3. The gate of the instantiated switch relation is given to the Test Execution component as an *abstract stimulus*. The term abstract indicates that the instantiated switch relation is an abstract representation of some computation steps taken in the SUT. For instance, a transition with label '?connect' is an abstract stimulus of the actual setup of a TCP connection between two distributed components of the SUT.
4. The translation of an abstract stimulus to a concrete stimulus is done by the Test Execution component. This component provides the stimulus to the SUT. When the SUT responds, the Test Execution component translates this response to an abstract response. For instance, the Test Execution component receives an HTTP response that the TCP connect was succesful. This is a concrete response, which the Test Execution component translates to an abstract response, such as a transition with label '!ok'. The Test Manager is notified with this abstract response.
5. The Test Manager translates the abstract response to an instantiated switch relation and updates the STS Engine. If this is possible according to the model, the Test Manager gives a pass verdict for this test. Otherwise, the result is a fail verdict.

2.5.2 GROOVE

GROOVE is an open source, graph-based modelling tool in development at the University of Twente since 2004 [23]. It has been applied to several case studies, such as model transformations and security and leader election protocols [6].

The architecture of the GROOVE tool is shown graphically in Figure 2.10. A graph grammar is given as input to the Rule Applier component, which determines the possible rule transitions. An Exploration Strategy can be started or the user can explore the states manually using the GUI. These components request the possible rule transitions and respond with the chosen rule transition (based on the exploration strategy or the user input). The Exploration Strategy can do an exhaustive search, resulting in a GTS. The graph states and rule transitions in this GTS can then be inspected using the GUI.

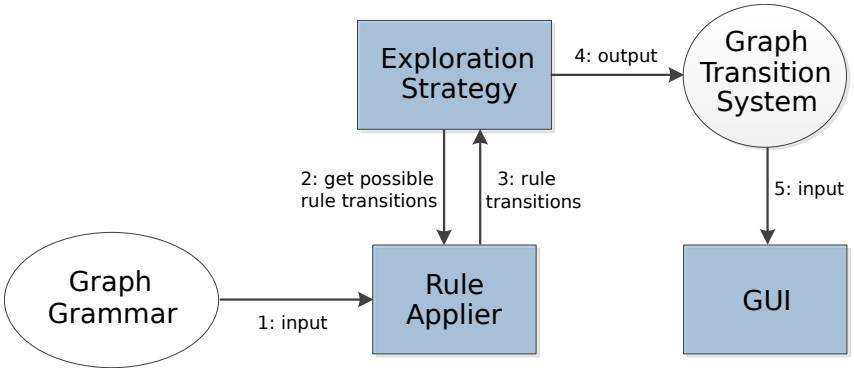


Figure 2.10: The GROOVE Tool

2.5.3 Graph grammars in GROOVE

The running example from Figure 2.4 is displayed as a graph grammar, as visualized in GROOVE, in Figure 2.11. The *LHS*, *RHS* and *NAC* of a rule in GROOVE are visualized together in one graph. Figures 2.11b, 2.11c and 2.11d show three rules. Figure 2.11a shows the start graph of the system.

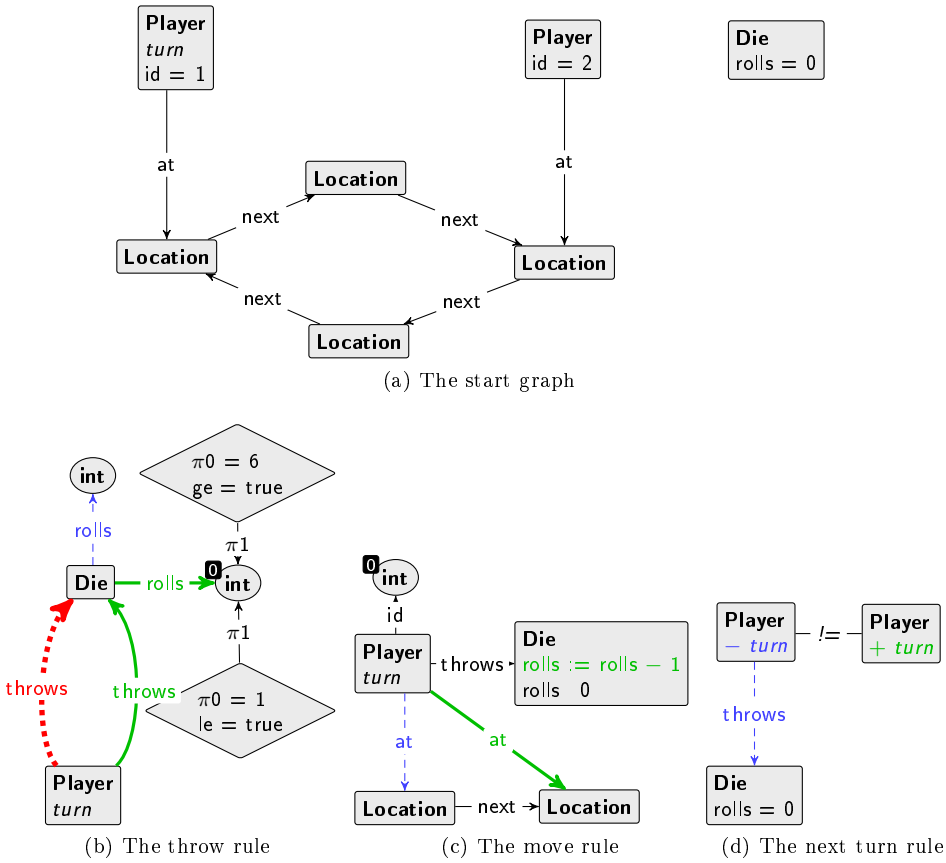
The colors on the nodes and edges in the rules represent whether they belong to the *LHS*, *RHS* or *NAC* of the rule.

1. normal line (black): This node or edge is part of both the *LHS* and *RHS*.
2. dotted line (red): This node or edge is part of the *NAC* only.
3. thick line (green): This node or edge is part of the *RHS* only.
4. dashed line (blue): This node or edge is part of the *LHS* only.

The rules can be described as follows:

1. 2.11b: 'if a player has the turn and he has not thrown the die yet, he may do so.'
2. 2.11c: 'if a player has the turn and he has thrown the die and this number is larger than zero, he may move one place and then it is as if he has thrown one less.'
3. 2.11d: 'if a player has finished moving (number thrown is zero), the next player receives the turn.'

The strings on the nodes are a short-hand notation. The bold strings, **Die**, **Player**, **Location** and **int** indicate the *type* of the node. Nodes with a type starting with a lower case letter, such as



(a) The start graph

(b) The throw rule

(c) The move rule

(d) The next turn rule

Figure 2.11: The graph grammar of the board game example in Figure 2.4

int, are variable nodes from $\mathcal{V}$. The italic string *turn*, is a representation of a self-edge with label *turn*. In the next turn rule, the *turn* edge exists in the *LHS* as a self-edge of the left **Player** node and in the *RHS* as a self-edge of the right **Player** node. In the same rule, the *throws* edge from the left **Player** node to an integer node only exists in the *LHS*.

The 'throws $>$ 0' is a term over the variable node that is the target of an outgoing edge labeled 'throws'. In this case, the valuation of the term be true for the rule to match the graph.

The number '0' in the top left of the **int** node in the throw rule indicates that this integer is the first parameter in $param(l)$, where l is the label on the rule transition created by applying the throws rule.

The graph is transformed after the rule is applied. The resulting graph after the transformation is the new state of the system and the rule is the transition from the old state (the graph as it was before the rule was applied) to the new state. Figure 2.12 shows the IOGTS of one *?throws* rule application on the start graph. Note that the *?throws* is an input, as indicated by the '?'. State s_1 is a representation of the graph in Figure 2.11a. Figure 2.13 shows the graph represented by s_2.

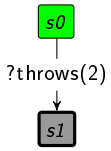


Figure 2.12: The GTS after one rule application on the board game example in Figure 2.11

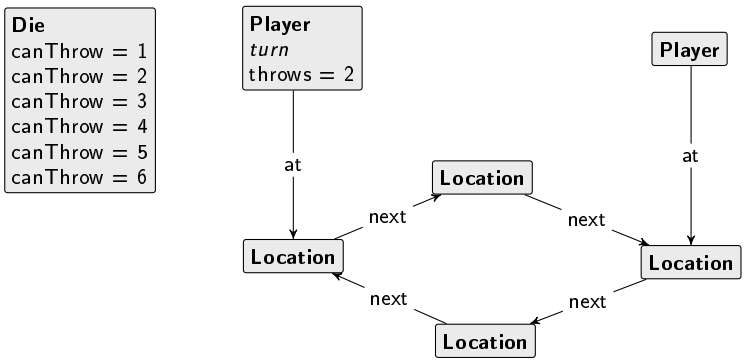


Figure 2.13: The graph of state $s2$ in Figure 2.12

Chapter 3

From Graph Grammar to STS

3.1 Requirements considerations

In order to do model-based testing with GGs, stimuli and responses have to be obtained from the GG. ATM uses an IOSTS, where the instantiated switch relations represent a stimulus to or a response from the SUT. To get an equivalent notion of stimuli/respones in GGs, the GG must be extended to an IOGG by indicating for each transformation rule whether it is of the input or output type. Then the IOGG can be explored to an IOGTS. The input/output rule transitions of the IOGTS can be used as the abstract stimuli and responses.

The second requirement for the design is the possibility to measure coverage statistics. The exploration of a GG can be done in two ways: *on the fly*, rule transitions are explored only when chosen by ATM, or *offline*, the GG is first completely explored and then sent to ATM. On-the-fly model exploration works well on large and even infinite models. However, coverage statistics cannot be calculated with this technique. The number of states (graphs) and rule transitions the model has when completely explored are not known, so a percentage cannot be derived. As coverage statistics are an important metric, the offline model exploration is chosen for GRATiS.

The last requirement is efficiency. An IOGTS can potentially be infinitely large, due to the range of data values. A model that is more efficient with data values is an STS. The setup of GRATiS is therefore to transform the IOGG directly to an IOSTS. Note that the first requirement is met, because locaton and switch relation coverage can be calculated on the IOSTS.

Taking these requirements into account, the method to achieve the goal of model-based testing on GGs is the following three steps:

1. Assign I/O types to graph transformation rules
2. Create an IOSTS from the IOGG
3. Perform the model-based testing on the IOSTS

This chapter describes an algorithm for creating an IOSTS from an IOGG.

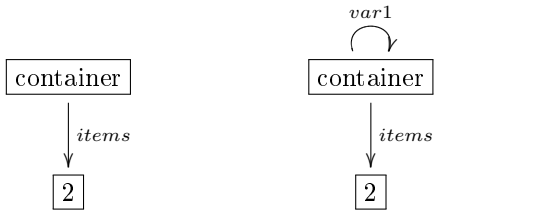
3.2 Point algebra

We define a *point algebra* $\mathcal{P}$ to be an algebra with $\forall s \in S.|\mathbb{U}^s_{\mathcal{P}}| = 1$. Each graph in $\mathcal{G}$ using the point algebra is structurally unique upto isomorphism; different values on value nodes are eliminated by the point algebra and two structurally equivalent graphs are the same graph. Therefore, using this

algebra is efficient when exploring the GTS. The loss of information is only in the concrete values at each state. This information is also not present in an STS, which treats the values symbolically as variables.

3.3 Variables

The variables in an STS represent an aspect of the modelled system. For instance, if a system keeps track of the number of items in containers, the STS modelling this system could have integer location variables $items_1..items_n$. The value nodes in a host graph are a representation of one element from the universe of elements of the same sort. Edges can exist between graph nodes and value nodes. The same example modelled in a graph grammar could be a graph node representing a container with an edge labelled 'items' to an integer node. This is shown in Figure 3.1a. This is a common way of representing a variable in a GG. Here the combination of edge plus source node represents the variable. However, the source node identity is not consistent through graph transformations, as the graphs are structurally unique upto isomorphism. In order to have variables in GGs, the source node must be made structurally unique, by means of a self-edge. Figure 3.1b shows the self-edge on the container node. The variable *var1_items*, the number of items in the container, is now represented by this graph.



(a) Common way (b) Persistent way

Figure 3.1: Possible ways of modelling variables in GGs

On the basis of the discussion above, we introduce the following terminology. The labels on the self-edges we call *variable labels*, represented by L_{var}. The edge having a variable label we call a *variable edge*. The source and target node of the variable edge we call the *variable anchor*.

3.4 The algorithm

Let $J = \langle W, w_0, \mathcal{L}, \imath, \mathcal{I}, \Lambda, D \rangle$ be an IOSTS and let $K = \langle R, G_0 \rangle$ be an IOGG. The first step in the algorithm is to explore the GG K using the point algebra $\mathcal{P}$ to an IOGTS $O_{\mathcal{P}} : \langle G, R, M, U, G_0 \rangle$.

3.4.1 Locations

The set of locations W is chosen to be equal to the set of graphs G. Additionally, the initial location w_0 is equal to the initial graph G_0.

3.4.2 Location variables

The location variables are a subset of the product of variable labels and regular labels, given by $\mathcal{L} \subset L_{var} \times L$. The set of location variables in a host graph $\langle V_{G^h}, E_{G^h} \rangle$ is defined by the following. $\langle l_{var}, l \rangle \in \mathcal{L}$ if:

- $\langle w \in \mathbb{W}, l, u \in \mathbb{U} \rangle \in E_{G^h}$ - the label must be on an edge from a graph node to a value node.
- $\langle w, l_{var}, w \rangle \in E_{G^h}$ - the variable label must be a self edge on the same graph node.

The initialization $\imath$ is then given by $\langle l_{var}, l \rangle \mapsto u$.

3.4.3 Gates

The gate of a switch relation represents the stimulus to or response from the SUT. In an IOGG, the rules are this representation. Therefore, the set of gates Λ is chosen to be equal to the set of rules R.

3.4.4 Interaction variables

Interaction variables are used by the gates to represent a stimulus or response variable. The variable nodes in rule graphs are this representation. The set of interaction variables $\mathcal{I}$ is chosen to be equal to the set of variable nodes $\mathcal{V}$. For a rule r and all variable nodes $\mathcal{V}_r$ in *LHS* of r, $arity(r) = |\mathcal{V}_r|$.

3.4.5 Guards

The guard of a switch relation restricts the use of the switch relation based on the values of the variables. In a GG, a rule is restricted by the terms. The variables used in the terms are interaction variables. Therefore, the first part of the guard is constructed by joining the terms for each term node by $\bigwedge_{z \in V_{G^r} \cap 2^{\mathcal{T}}} \bigwedge_{t_1, t_2 \in z} t_1 = t_2$. Using a rule match m, the second part is constructed. For a $LHS = \langle V_{G^r}, E_{G^r} \rangle$ the smallest set of terms T, such that $\langle m(z), l \rangle = x \in T$ when:

- $x \in \mathcal{V} \cap V_{G^r}$
- $\langle z, l, x \rangle \in E_{G^r}$
- $m(z)$ is a variable anchor

Then, the terms are joined by $\bigwedge_{t \in T} t$.

3.4.6 Update mappings

An edge with label l from a variable anchor z to a value node can be erased from the graph and a new edge with label l from z to a new value node can be created by a rule. This inidicates an update for the location variable given by $\langle z, l \rangle$. In the rule graph, the *RHS* of the rule has the pre-image of the z and the edge to a variable node, given by the interaction variable x. The update mapping for this example is: $\langle z, l \rangle \mapsto x$.

3.4.7 Switch relations

A rule transition $G \xrightarrow{r,m} G' \in U$ is mapped to a switch relation $(G \xrightarrow{r,\gamma,\rho} G') \in D$. The guard and update mapping are constructed according to sections 3.4.5 and 3.4.6 using r and m.

3.5 Constraints

This section describes the constraints on the algorithm in section 3.4.

3.5.1 Constraint 1: unique variables

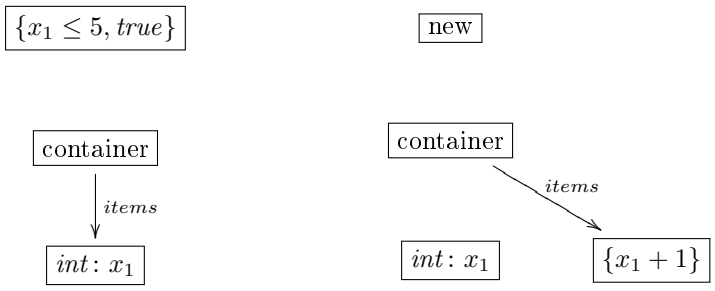
A location variable is indicated by a node and label pair $\langle z, l \rangle$. This pair must be unique, i.e. no two edges $\langle z, l, z' \rangle, \langle z, l, z'' \rangle$ may exist where $z' \neq z''$. Otherwise, it is possible that a variable has two different values.

3.5.2 Constraint 2: no variables in NACs

Let $\langle z, l, v \in \mathcal{V} \rangle$ be an edge in a rule graph in the NAC of a rule. Let $\{v, 1\}$ be a term node in the same rule graph. This is a common way of expressing that the v node may not have the value 1 as image. However, using the point algebra this rule will never match, because there is only one possible image for the variable node and the value 1 in the point algebra. A correct way of modelling this example, is having the term node $\{v = 1, false\}$ in the *LHS* of the rule. In the point algebra, both terms valuate to the same boolean value and an image for this term node can always be found.

3.5.3 Constraint 3: structural constraints on node creating rules

In the previous constraint it is shown that a term node in the *LHS* always has an image, if all terms are of the same sort. Figure 3.2 shows the *LHS* and *RHS* of a rule in the container-items example. The rule adds an item to the container unless it is full, i.e. has five items. If an item is added, a new node is created in the host graph. Using the point algebra, this rule creates an infinite number of structurally unique graphs. Therefore, the exploration never ends. Node creating rules must have structural constraint(s), such that an infinite exploration is prohibited.



(a) LHS (b) RHS

Figure 3.2: Node creating rule without structural constraint

Chapter 4

Implementation

4.1 General setup

GRATiS uses GROOVE as a replacement of the IOSTS in ATM. Figure 4.1 shows the class diagram of GRATiS. GROOVE has several exploration strategies for exploring a GG to a GTS. The GROOVE interface is added to replace the exploration strategy. It contains functionality to build an STS from a GG and send an STS to a remote host. This is the most logical place in GROOVE, because the exploration strategy explores the GTS, which is needed to build the STS. On ATM, a component is added that receives an STS and starts a test run.

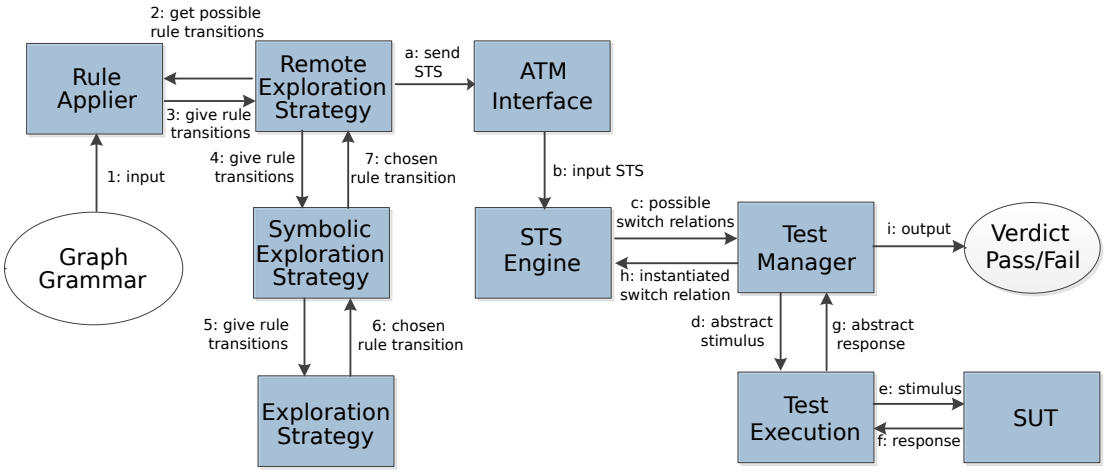


Figure 4.1: The GRATiS class diagram

4.2 Description of added functionality

This section covers in detail the added functionality to GROOVE and ATM.

4.2.1 GROOVE Interface

Figure 4.2 shows the class diagram of the added exploration strategy interface. The symbolic exploration strategy has an exploration strategy such as the Breadth-First exploration strategy to

explore the GTS. The remote exploration strategy extends the symbolic exploration strategy.

The user starts the remote exploration strategy. This strategy starts a Breadth-First exploration strategy. This strategy explores the GTS and notifies the remote strategy when there are no more rule transitions to explore. The symbolic strategy builds the STS in Java objects using the explored rule transitions. The class diagram of the STS is given in section 4.2.2. The remote strategy sends the STS in JSON format as a HTTP PUT request to the interface at ATM.

The Breadth-First exploration strategy and the symbolic strategy are loosely coupled, such that other strategies can also be used if desired.

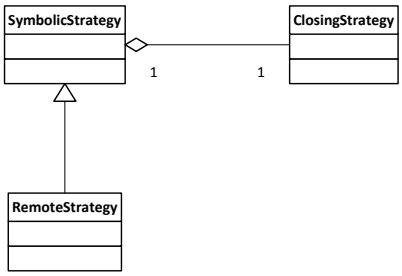


Figure 4.2: The class diagram of the exploration strategy interface

4.2.2 STS

Figure 4.3 shows the class diagram of the STS in GRATiS. The STS is composed of Locations, Switch relations, Gates, Interaction and Location variables. A Location can be the start and targent of any number of switch relations. A switch relation has two locations; the start and target location. A Switch relation has one gate and a gate can belong to any number of switch relations. A gate can have any number of interaction variables, but an interaction variable belongs to one gate. The STS has a singleton class, the RuleInspector, which contains the functionality of building guards and updates from rule graphs.

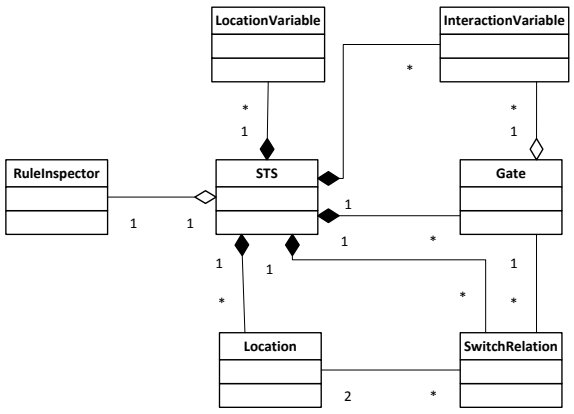


Figure 4.3: The class diagram of the STS in GRATiS

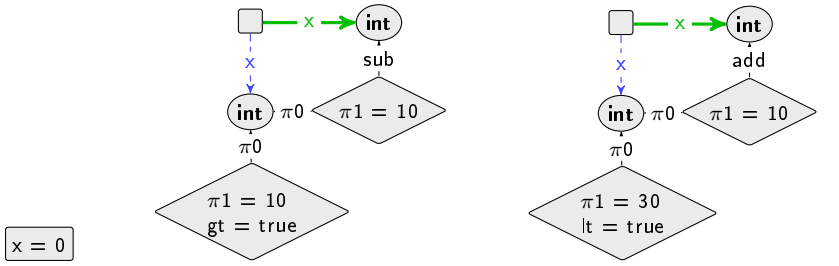
4.2.3 ATM Interface

The ATM interface is one component in the Rails framework. It receives the STS request, builds the STS as Ruby objects and initiates the test run using this STS as model.

4.3 Rule priority

This section covers a specific implementation issue of setting rule priorities in GROOVE.

There can be several outgoing rule transitions from a graph. In GROOVE, rules can have different priority levels. A rule transition with a higher priority rule is explored before rule transitions with lower priority rules. Consider the graph grammar in Figure 4.4. The 'add' rule produces a rule transition to a graph, where the *sub* rule produces a rule transition back to the start graph. The 'sub' rule does not match the start graph, because it has a lower priority than the *add* rule.



(a) The initial graph

(b) The *sub* rule with priority 0

(c) The *add* rule with priority 1

Figure 4.4: Priority rules

The graphs are isomorphic under the point algebra, so they represent the same location. The STS of transforming this graph grammar is in Figure 4.5, with $\imath = \{x \mapsto 25\}$. This STS is wrong, because the 'sub' switch relation can be taken from the start.

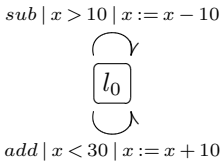


Figure 4.5: A wrong STS transformation of the graph grammar in Figure 4.4

The solution is shown in Figure 4.6. The negated guard of the 'add' switch relation is added to the 'sub' switch relation. The optimized guard for this switch relation is 'x >= 30' of course, but this shows the main principle: for each outgoing switch relation, the negated guard of all switch relations represented by higher priority rules must be added to the guard. So, the 'x < 30' guard is negated to '!(x < 30)' and added to yield the 'x > 10 && !(x < 30)' guard. Note that if the 'add' switch relation had no guard, it would be applicable on all graph states with isomorphic abstractions. Therefore, the 'sub' switch relation would not exist, because the 'add' rule is always applicable whenever the 'sub' rule also is.

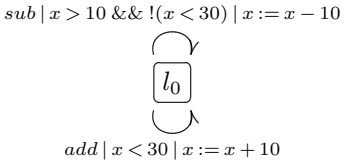


Figure 4.6: A correct STS transformation of the graph grammar in Figure 4.4

Chapter 5

Validation

This chapter covers the techniques used for the validation of the design. The validation is done through examples and a case-study, reported in section 5.1. Possible measurements on models and the performance are reported in section 5.2.

5.1 Validation models

Several examples of small systems are used to validate GRATiS. Each example is modelled using a GG and an STS. The examples are:

- a boardgame - a game with a stochastic element
- the farmer-wolf-goat-cabbage puzzle - a datafree puzzle with states
- a reservation system - a system where people can make reservations for tables at a restaurant
- a bar tab - a system where people order drinks on their tab

In addition, a case study is done where GRATiS is tested on a larger software system. The system is a *self-scan register*, which allows customers of a supermarket to scan and pay for their products without help of an employee. The communication protocol between these two units is modelled using an IOGG, based on the specification of the system. This model is used for the model-based testing against the live system.

5.2 Measurements

This sections lists possible measurements on the models and execution of GRATiS on those models.

5.2.1 Simulation and redundancy

The STS created by GRATiS and the created STS by hand can be compared. It can be observed whether the STS created by GRATiS simulates the STS created by hand and vice versa. When either is not the case, the models show a different possible behaviour for the SUT. The reasons behind this difference are then explored.

It is possible that the generated STS s is larger than the STS built by hand t, even if both simulate each other. It is measured whether $traces(s) \subseteq traces(t) \wedge ((|D_t| < |D_s| \wedge |\mathcal{L}_t| <= |\mathcal{L}_s|) \vee (|D_t| <= |D_s| \wedge |\mathcal{L}_t| < |\mathcal{L}_s|)$. This indicates the the STS s is redundant.

5.2.2 Performance

The performance in terms of runtime and heap-size can be measured and compared. Assuming both the STS created by GRATiS and by hand simulate each other, these metrics will be the same for the testing part. Therefore, the runtime and heap size of the STS creation is measured.

5.2.3 Model complexity

The complexity of the generated STS and the GG can be measured using Halstead's software science [7]. This method is used in measuring software complexity, but can also be used in analyzing model complexity. In Halstead's software science, the operators and operands are counted. The operators are the function symbols, the operands are the identifiers. Both models have nodes and edges. In an STS, the locations are counted as nodes, the switch relations as edges. Nodes and edges are considered to be operands. The distinct number of operators (n_1) and operands (n_2) are counted as well as the total number of operator occurrences (N_1) and operand occurrences (N_2). These metrics combined lead to the *Volume* of the models. The volume is calculated by: $(N_1 + N_2) * log_2(n_1 + n_2)$. Comparing the volumes of the STS and GG gives an indication of the relative model complexity.

5.2.4 Extendability

The models can be extended to include more functionality. In this measurement, a realistic scenario is introduced where additional functionality is required. It is then measured how much the complexity increases, using the measurment in section 5.2.3.

Chapter 6

Model Examples

This chapter contains a GG and STS for each example in section 5.1 and the measurements for each pair.

6.1 Example 1: boardgame

The boardgame is the running example of which the IOSTS and IOGG are given in Figures 2.4 and 2.11 respectively.

6.1.1 Simulation and redundancy

The responses used by the IOSTS and by the IOGG are different. Both models, used as examples to clarify the IOSTS and IOGG formalisms, were built with a different behavior in mind. Both allow a die to be thrown, after which the IOSTS directly moves the player to the correct location and passes the turn and the IOGG moves the player by a series of responses ended with a $!nextTurn$. Therefore, both IOSTSs do not simulate each other.

6.1.2 Performance

The IOSTS is generated in a runtime of 300 ms using a heap-size of 1.9 MB.

6.1.3 Model complexity

start: 13 distinct operands 1 distinct operator 33 operands 3 operators

move: 2 new distinct operands 5 new distinct operators 27 operands 6 operators

nextTurn: 0 new distinct operands 1 new distinct operator 13 operands 5 operators

throws: 3 new distinct operands 2 new distinct operators 30 operands 10 operators

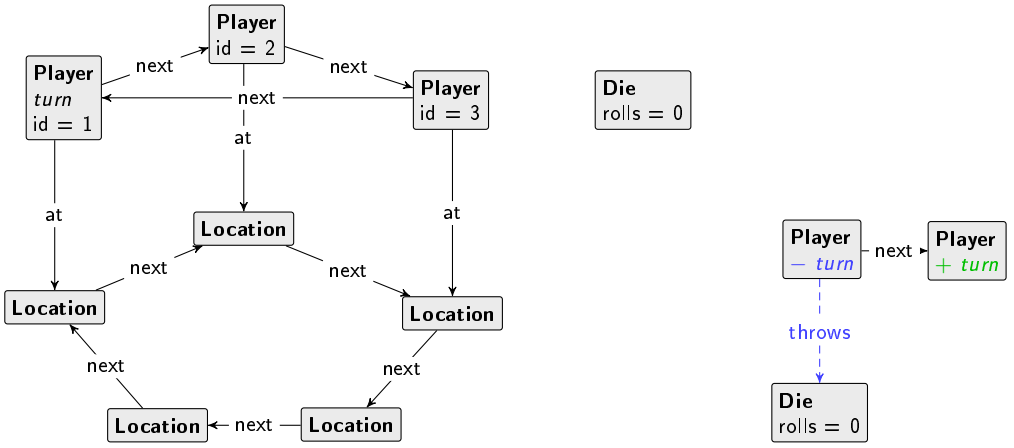
$n_1 = 9, n_2 = 18, N_1 = 24, N_2 = 103$ Volume is 127*4.75 = 603.25

IOSTS: 22 distinct operands 5 distinct operators 62 operands 25 operators

$n_1 = 5, n_2 = 22, N_1 = 25, N_2 = 62$ Volume is 87*4.75 = 413.25

6.1.4 Extendability

The boardgame is extended to include more players and locations. For the IOGG, this means adding new locations and players to the initial graph. The players get a fixed order in which they play. This means that the next turn rule also has to be extended. The result is in Figure 6.1. This extension reduces the distinct number of operators by 1 and introduces no new operands. The number of operator occurences has decreased by 1 and the number of operand occurences has grown by 10.



(a) The initial graph

(b) The next turn rule

Figure 6.1: The extended graph grammar of the board game example in Figure 2.11

The IOSTS gains a variable and a switch relation for the new player. The result is in Figure **??**. The distinct number of operators has not increased and the distinct number of operands has increased by 1. The number of operator occurences has increased by 9 and the number of operand occurences has increased by 17.

The volume of the IOGG has increased by 35.95. $n_1 = 8, n_2 = 18, N_1 = 23, N_2 = 113$ Volume is 136*4.70 = 639.20 The volume of the IOSTS has increased by 130.28. $n_1 = 5, n_2 = 23, N_1 = 34, N_2 = 79$ Volume is 113*4.81 = 543.53

6.2 Example 2: farmer-wolf-goat-cabbage

In this puzzle, a farmer, wolf, goat and cabbage are on one side of a river. The farmer can take upto one object to the other side. If the wolf and goat are on one side of the river without the farmer, the wolf eats the goat and the puzzle is reset. This also holds for the goat and the cabbage. The goal is to move all four to the other side of the river. The IOGG of this puzzle is in Figure 6.2. Here the 'move' and 'invalid' rules are similar, therefore only the 'move cabbage' rule is shown. The response rules '!retry', '!eaten' and '!done' have a higher priority. This ensures that a proper response is given after a move, before allowing more stimuli. The IOSTS of this puzzle is in Figure 6.3.

6.2.1 Simulation and redundancy

Both the generated IOSTS and the IOSTS built by hand allow all inputs and give the appropriate responses when necessary. This shows that both IOSTSs simulate each other. The generated IOSTS has 50 switch relations and 0 location variables. The IOSTS built by hand has 11 switch

relations and 4 location variables. The IOGG does not use variables to track the location of each item. Therefore the generated IOSTS has a location per state of the puzzle.

6.2.2 Performance

The IOSTS is generated in a runtime of 770 ms using a heap-size of 5.2 MB.

6.2.3 Model complexity

6.2.4 Extendability

In another variant of this puzzle, when one of the objects is eaten, the puzzle does not reset but undoes the last action. Figure 6.4 shows this extension in two rules: the 'move cabbage' and the 'eaten undo' rule. The rules keep track of the last moved items. When an item gets eaten, the last move can be undone.

6.3 Example 3: restaurant reservations

Figure 6.5a shows the initial graph of three tables at a restaurant and two potential customers. Figure 6.5b shows part of a rule that allows people to make reservations. The start and end times are timestamps represented by integers. This rule allows people to make multiple reservations. However, this rule violates the constraint in section 3.5.1, because the reservation objects are not unique. Allowing a dynamic amount of reservations per person means that variables need to be introduced dynamically as well or more complex variables have to be used, such as arrays. To model this system using an IOSTS, arrays are also needed.

6.4 Example 4: bar tab system

This example models a bar tab system, where customers can order beer, wine and soda. The price of the order adds to tab. Customers can pay their tab with money; they receive cash back if the payment exceeds the tab. The model is abstracted to include three customers. Furthermore, a customer can order only one drink. Drinks and payments are processed immediately before other drinks or payments can occur. The stimuli accepted by the system are $?o(i,d), ?p(i,p)$ for ordering a drink d on bar tab i and paying amount p on bar tab i respectively. The responses by the system are $!po(b), !pp(b,r)$ for processing an order giving the new bar tab balance b and processing a payment giving the new account balance b and the return funds r respectively.

Figure 6.6 shows the IOGG of the bar tab system. The '!process_order' and '!process_payment' rule have a higher priority than the '?order' and '?pay' rule. Figure 6.7 shows the IOSTS of the bar tab system. The IOSTS uses the variables T_1, T_2, T_3 to keep track of the bar tabs of the three people. It uses the variables I, P as temporary variables for the id and payment/price respectively. The function m takes the maximum value of its parameters.

6.4.1 Simulation and redundancy

Both the generated IOSTS and the IOSTS built by hand correctly allow the ordering of drinks and payment of bar tabs. This shows that both IOSTSs simulate each other. The generated IOSTS has 24 switch relations and 13 location variables. The IOSTS built by hand has 10 switch relations

and 5 location variables. This shows that the generated IOSTSs is redundant. The first IOSTS keeps the name and price of drinks as location variables, whereas the latter IOSTS hard-codes these into the guards and updates. The generated IOSTS builds a switch relation with gate $?o$ for each combination of customer and drink. It also builds a switch relation with gate $?p$ for each customer. The target locations of all these switch relations have one switch relation back to the initial location. Therefore, the number of switch relations is $3 * 3 * 2 + 3 * 1 * 2 = 24$.

6.4.2 Performance

The IOSTS is generated in a runtime of 250 ms using a heap-size of 2.1 MB.

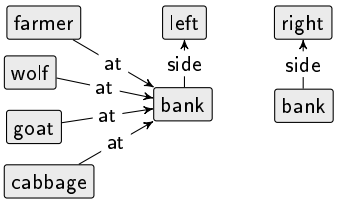
6.4.3 Model complexity

6.4.4 Extendability

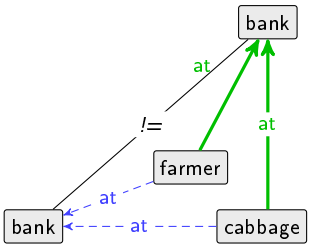
The system is extended to allow ordering multiple drinks of different types. Also, a customer can purchase the option of receiving 10% discount on all ordered drinks for 50 euros (added to the tab). The stimulus given is $?d$ and the response is $!pd(b)$ where b is the new balance.

6.5 Conclusions

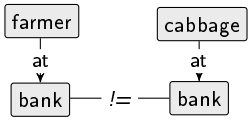
Need more complex data structures. And it is possible! runtime and heap-size: negligible. boardgame: different approaches in formalisms if requirements are not set. Note that abstract stimuli/responses are translated into non-abstract stimuli by Test Execution module. A response notifying player x has moved to location y can be translated to a serie of abstract responses. Advantage: allows model to be more flexible, have better design. Disadvantage: less clear according to specification, also more work in translating abstract stimuli/responses. fwgc: 50 locations doable, no need to look at the generated STS. 11 switches and 4 variables easier visualized than 50 switches. Is this true? bartab: more logical placement of concerns. tab is increased and deducted at the process rules, sts sets price of drink during ordering. IOGG uses OO by grouping names and prices and tabs and amounts.



(a) The initial graph

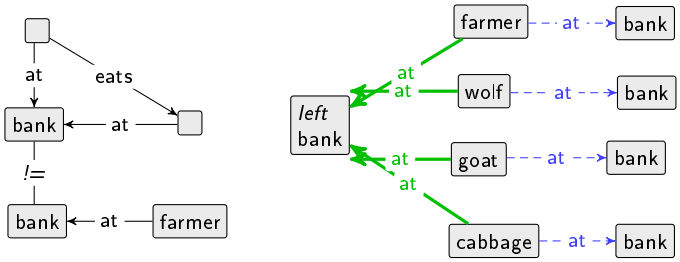


(b) The ?c (move cabbage) rule with priority 0

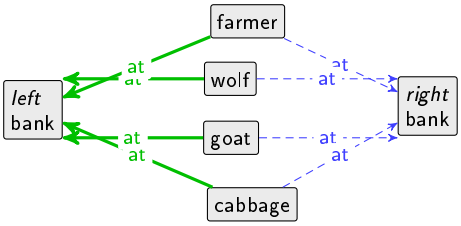


(c) The ?c (invalid cabbage move) rule with priority 0

(d) The !retry rule with priority 1



(e) The !eaten rule with priority 1



(f) The !done rule with priority 1

Figure 6.2: The graph grammar of the farmer-wolf-goat-cabbage puzzle

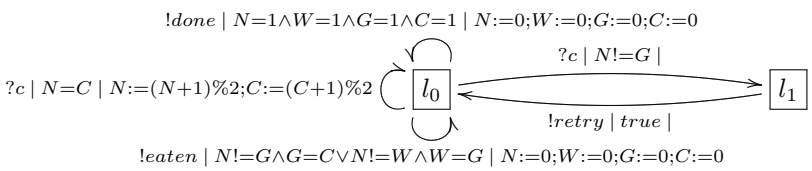
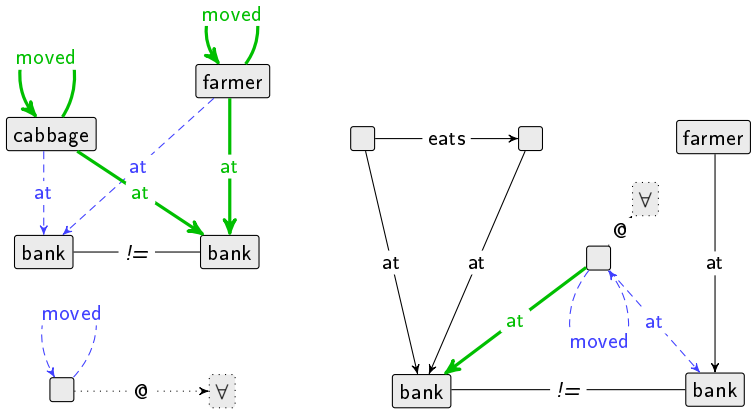
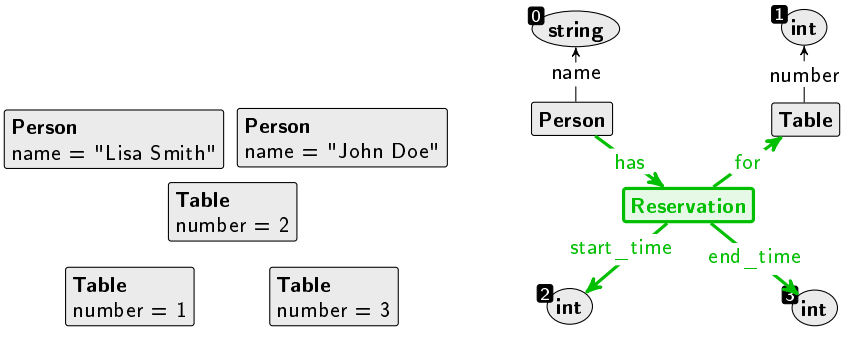


Figure 6.3: The IOSTS of the farmer-wolf-goat-cabbage puzzle



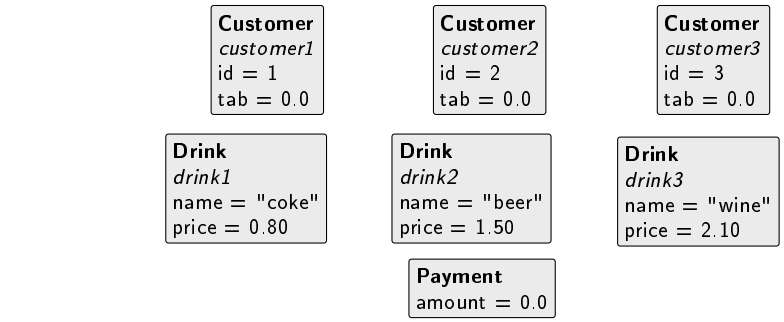
(a) The move cabbage rule (b) The eaten undo rule

Figure 6.4: The extended graph grammar of the board game example in Figure 2.11



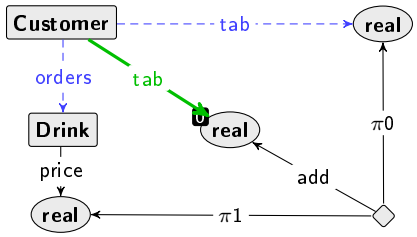
(a) The initial graph (b) The make reservation rule

Figure 6.5: The graph grammar of the restaurant reservation system

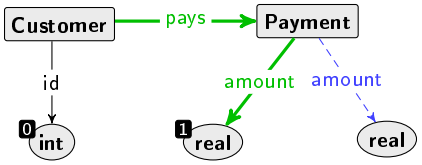


(a) The initial graph

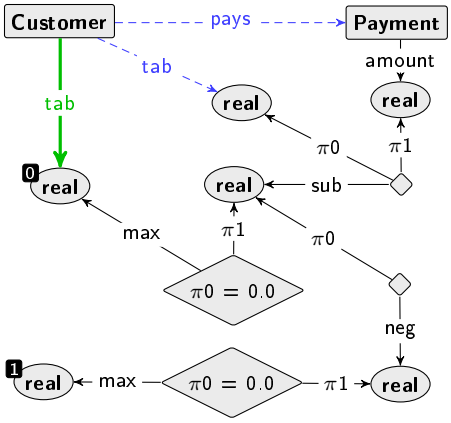
(b) The ?o rule with priority 0



(c) The !po rule with priority 1



(d) The ?p rule with priority 0



(e) The !pp rule with priority 1

Figure 6.6: The graph grammar of the bar tab system

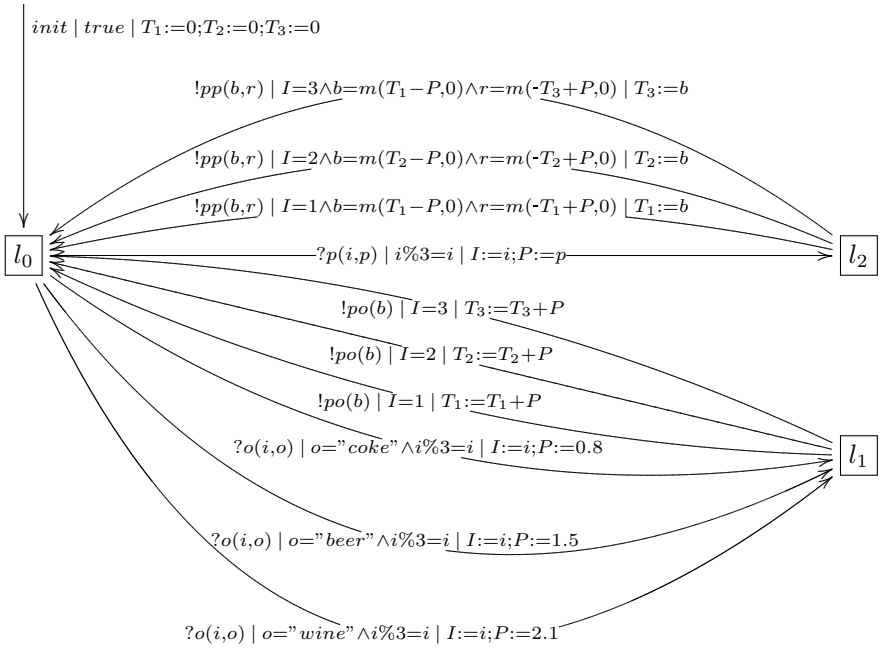


Figure 6.7: The IOSTS of the bar tab system

Chapter 7

Case Study

This chapter gives the models and measurements made for the Self-Scan Register Protocol (SCRP) case study.

7.1 Scanflow Cash Register Protocol

The system used for this case study is a *self-scan register*, which allows customers of a supermarket to scan and pay for their products without help of an employee. Figure 7.1 shows this self-scan register. The system contains a *scan-flow unit*, which scans the products, and a *cash register unit*, which allows for the payment. The communication protocol between these two units is modelled as a GG, shown in Figure **??**.

Figure 7.1: A self-scan register

7.2 Measurements

Show the measurements done on the GG and STS of SCRP.

Chapter 8

Conclusion

8.1 Summary

8.2 Conclusion

8.3 Future work

- bedachte measurements die niet gedaan zijn - on the fly model generation implementeren - complexere datastructuren (set, map, array)

ACKNOWLEDGEMENTS

Bibliography

[1] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1 – 54, 1999.

[2] Axel Belinfante. JTorx: A tool for on-line model-driven test derivation and execution. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer Berlin / Heidelberg, 2010.

[3] E. Brinksma. A theory for the derivation of tests. *Protocol Specification, Testing, and Verification VIII*, 1988.

[4] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A symbolic test generation tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 151–173. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-46002-0_34.

[5] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton. The aetg system: an approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on*, 23(7):437 –444, jul 1997.

[6] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using GROOVE. *International journal on software tools for technology transfer*, 14(1):15–40, February 2012.

[7] M.H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[8] Hasan and Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311 – 325, 1992.

[9] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1 SPEC. ISS.):187–198, 2006.

[10] M.R.A. Huth and M. Ryan. *Logic in computer science: modelling and reasoning about systems*. Cambridge University Press, 2000.

[11] C. Laird, L. & Brennan. *Software measurement and estimation: A practical approach*. IEEE Computer Society/Wiley, 2006.

[12] Tim A.C. Willemse Lars Frantzen, Jan Tretmans. Test generation based of symbolic specifcations. Technical report, Nijmegen Institute for Computing and Information Sciences (NIII), 2005.

[13] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090 –1123, aug 1996.

[14] R. Busser M. Blackburn and A. Nauman. Why model-based test automation is different and what you should know to get started. *International Conference on Practical Software Quality and Testing*, 2004.

[15] Joost-Pieter Katoen Manfred Broy, Bengt Jonsson and Alexander Pretschner. *Model-Based Testing of Reactive Systems.* Springer-Verlag Berlin Heidelberg, 2005.

[16] Thomas J. McCabe. Structured testing: A software testing methodology using the cyclomatic complexity metric. *National Bureau of Standards, Special Publication*, 1982.

[17] Steve McConnell. Software quality at top speed. *Softw. Dev.*, 4:38–42, August 1996.

[18] Lev Nachmanson, Margus Veanes, Wolfram Schulte, Nikolai Tillmann, and Wolfgang Grieskamp. Optimal strategies for testing nondeterministic systems. *SIGSOFT Softw. Eng. Notes*, 29:55–64, July 2004.

[19] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83 – 133, 1984.

[20] Martin Pol. *Testen volgens Tmap (in dutch, Testing according to Tmap).* Uitgeverij Tutein Nolthenius, 1995.

[21] S.J. Prowell. JUMBL: a tool for model-based statistical testing. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, page 9 pp., jan. 2003.

[22] A. Rensink. Towards model checking graph grammars. In S. Gruner and S. L. Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS), Southampton, UK*, volume DSSE-TR-2003-02 of *Technical Report*, pages 150–160, Southampton, 2003. University of Southampton.

[23] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485, Berlin, 2004. Springer Verlag.

[24] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-40911-4_20.

[25] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[26] Jan Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[27] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer Berlin / Heidelberg, 1996. 10.1007/3-540-61042-1_42.

[28] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, pages n/a–n/a, 2011.

[29] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In Robert Hierons, Jonathan Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer Berlin / Heidelberg, 2008.

[30] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.

List of Symbols